Abstract Data Types

An abstract data type (ADT) consists of:
- a mathematical model of the data
- methods for accessing and modifying the data

Example
The Dictionary ADT of Practical 5.

Example: Stacks and Queues

A Stack is an ADT for storing a collection of elements that supports the following methods:
- `push(c)`: Insert element c.
- `pop()`: Remove the most recently inserted element and return it; an error occurs if the stack is empty.
- `isEmpty()`: Return TRUE if the stack is empty and FALSE otherwise.

A Queue is an ADT for storing a collection of elements that supports the following methods:
- `enqueue(c)`: Insert element c.
- `dequeue()`: Remove the element inserted the longest time ago and return it; an error occurs if the queue is empty.
- `isEmpty()`: Return TRUE if the queue is empty and FALSE otherwise.

Both Stack and Queue can easily be realised by data structures based on arrays or linked lists.

Data Structures

A data structure realising an ADT consist of:
- collections of variables for storing the data described by the mathematical model underlying the ADT
- algorithms for the methods of the ADT.

Remark

<table>
<thead>
<tr>
<th>ADT</th>
<th>JAVA interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>data structure</td>
<td>JAVA class</td>
</tr>
</tbody>
</table>

Sequential Data

Mathematical model of the data: a linear sequence of elements.
- A sequence has well-defined first and last elements.
- Every element of a sequence except the first and last has a unique predecessor and successor.
- The rank of an element `c` in a sequence `S` is the number of elements before `c` in `S.`
Arrays and Linked Lists in Memory

An array, a singly linked list, and a doubly linked list storing objects o1, o2, o3, o4, o5:

Arrays and Linked Lists Abstractly

An array, a singly linked list, and a doubly linked list storing o1, o2, o3, o4, o5:

Vectors

A Vector is an ADT for storing a sequence S of n elements that supports the following methods:

- `elemAtRank(r)`: Return the element of rank r; an error occurs if r < 0 or r > n

- `replaceAtRank(r, c)`: Replace the element of rank r with c; an error occurs if r < 0 or r > n

- `insertAtRank(r, c)`: Insert a new element c at rank r (this increases the rank of all following elements by 1); an error occurs if r < 0 or r > n

- `removeAtRank(r)`: Remove the element of rank r (this reduces the rank of all following elements by 1); an error occurs if r < 0 or r > n

- `size()`: Return n, the number of elements in the sequence

An Array Based Data Structure for Vector

Variables

- Array A (storing the elements)
- Integer n = number of elements in the sequence
An Array Based Data Structure for Vector (cont’d)

Methods

**Algorithm elemAtRank[r]**
1. return $A[r]$

**Algorithm replaceAtRank[r, c]**
1. $A[r] \leftarrow c$

**Algorithm insertAtRank[r, c]**
1. for $i \leftarrow n$ downto $r + 1$ do
2. $A[i] \leftarrow A[i - 1]$
3. $A[r] \leftarrow c$
4. $n \leftarrow n + 1$

**Algorithm size[]**
1. return $n$

(insertAtRank assumes that array $A$ is big enough)

Abstract Lists

List is an ADT for storing a sequence that supports the following methods:
- $first()$: Return the position of the first element; an error occurs if the list is empty.
- $isEmpty()$: Return TRUE if the list is empty and FALSE otherwise.
- $next(p)$: Return the position of the element following the one at position $p$; an error occurs if $p$ is the last position.
- $isLast(p)$: Return TRUE if $p$ is the last position of the list and FALSE otherwise.
- $replace(p, c)$: Replace the element at position $p$ with $c$.
- $insertFirst(c)$: Insert $c$ as the first element of the list.
- $insertAfter(p, c)$: Insert element $c$ after position $p$.
- $remove(p)$: Remove the element at position $p$.
Plus: $last()$, $previous(p)$, $isFirst(p)$, $insertLast(c)$, and $insertBefore(p, c)$

Realising List with Doubly Linked Lists

Variables
- Positions of a List are realised by nodes having fields element, previous, next.
- The whole list is accessed through two node-variables first and last.

Methods (Two Examples)

**Algorithm insertAfter(p, c)**
1. create a new node $q$
2. $q\text{-element } \leftarrow c$
3. $q\text{-next } \leftarrow p$
4. $q\text{-previous } \leftarrow p$
5. $p\text{-next } \leftarrow q$

**Algorithm remove(p)**
1. $p\text{-previous, next } \leftarrow p\text{-next}$
2. $p\text{-next, previous } \leftarrow p\text{-previous}$
3. delete $p$

Sequences

Sequence is an ADT for sequences combining the methods of Vector and List. It can be realised by data structures using arrays or linked list.

Running times

<table>
<thead>
<tr>
<th>method</th>
<th>array</th>
<th>doubly linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td>$size, isEmpty, element, first, last, next, previous, replace, isFirst, isLast, insertLast$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$elemAtRank, replaceAtRank$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$insertFirst, insertAfter, insertBefore$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$insertAtRank, removeAtRank$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
**Dynamic Arrays**

VeryBasicSequence is an ADT for storing a sequence that supports the following methods:
- `elemAtRank(r)`: Return the element of S with rank r; an error occurs if \( r < 0 \) or \( r > n \).
- `replaceAtRank(r, c)`: Replace the element of rank \( r \) with \( c \); an error occurs if \( r < 0 \) or \( r > n \).
- `insertLast(c)`: Append element \( c \) to the sequence.
- `size()`: Return \( n \), the number of elements in the sequence.

**Dynamic Insertion**

**Algorithm insertLast(c)**

1. if \( n < A.length \) then
2. \( A[n] \leftarrow c \)
3. else \( n = A.length \), i.e., the array is full
4. \( N \leftarrow 2(A.length + 1) \)
5. Create new array \( A' \) of length \( N \)
6. for \( i = 0 \) to \( n - 1 \) do
7. \( A'[i] \leftarrow A[i] \)
8. \( A'[n] \leftarrow c \)
9. \( A \leftarrow A' \)
10. \( n \leftarrow n + 1 \)

**Analysis**

**Observation**

The worst case running time of `insertLast` is \( \Theta(n) \).

**Theorem**

Inserting \( m \) elements into an initially empty VeryBasicSequence using the Algorithm `insertLast` requires time \( O(m) \).

Thus the amortised running time of one insertion is \( \Theta(1) \).